Wstęp

Aplikacja **bluebox** umożliwia pobierania („kupowanie") produktów, które znajdują się w stanach magazynowych. Dostęp do serwisu przewidziany jest dla zarejestrowanych użytkowników (istnieje możliwość założenia nowego konta użytkownika – klienta). W ramach projektu są również nadane dwa dostępy, jako:

- Administrator systemu (login: **marek@trocha.net.pl**, hasło: **123**)
- Użytkownik, klient (login: **adam@nowak.pl**, hasło: **asc**)

Administrator konta posiada dostęp do wszystkich produktów wraz z możliwością edycji m.in. nazwy produktu, jego opisu, zmiany ilości sztuk w magazynie, zmiany ceny, itp, jak i usunięcia całego produktu. Ponadto, posiada również dostęp do wszystkich produktów, które zostały zakupione przez poszczególnych użytkowników (klientów). Konto standardowego użytkownika (klienta) jest ograniczone do wyświetlenia dostępnych produktów oraz produktów, które już klient „zakupił". Aplikacja została stworzona z wykorzystaniem:

- PHP 7.3
- Laravel 8.12
- Composer 2.0.4
- doctrine/dbal 3.0,
- Toolbar debugbar 3.5
- Moduł autoryzacji (laravel/ui 3.1)
- IDE: Visual Studio Code
- Bootsrap (w tym jquery)
- architekturą MVC

W razie pytań, sugestii zapraszam do kontaktu:

Marek Trocha
tel. 693 877 101
e-mail: marek@trocha.net.pl
strona: trocha.net.pl

Instrukcja stworzenia aplikacji

I tu będzie dalszy ciąg opisu aplikacji, w tym odniesienie do architektury MVC. Etapy tworzenia aplikacji zostały przedstawione w poniższej tabeli:

Etap	Opis / szczegóły
1.	Przygotowanie środowiska programistycznego a. Utworzenie katalogu **cs** i zainstalowanie w nim frameworka Laravel Komenda: `composer create-project --prefer-dist laravel/laravel bluebox` b. Przejście do wcześniej utworzonego projektu (katalogu) **bluebox** Komenda: `cd bluebox` c. Start serwisu po wywołaniu poniższej komendy i uruchomienie http://127.0.0.1:8000 Komenda: `php artisan serve`
2.	Doinstalowanie pakietów (szczegóły w pliku composer.json) a. Instalacja pakietu autoryzacji użytkowników Komenda: `composer require laravel/ui` oraz `php artisan ui:auth` b. Instalacja pakietu, odpowiedzialnego za ujednolicanie kolumn w bazie danych: doctrine/dbal Komenda: `composer require doctrine/dbal` c. Instalacja narzędzia developerskiego debugbar Komenda: `composer require barryvdh/laravel-debugbar –dev`

3.	Tworzenie podstrony serwisu bluebox z nazwą *products*
	a. **Tworzenie kontrolera** a w niej metody, odpowiedzialnej za wyświetlenie zawartości
	Komenda: `php artisan make:controller ProductController`
	`public function index() {`
	`return view('products.list');` - więcej w części c
	`}`
	b. **Dodanie routingu** w pliku web.php
	Route::get('products/', 'App\Http\Controllers\ProductController@index');
	c. **Tworzenie widoku:** resources\views\products\list.blade.php z biblioteką Bootsrap.
	▪ Przeniesienie części stałej serwisu do nowego pliku template.blade.php (katalog wyżej) utworzonego również w katalogu *products* – **@yield('content')**
	▪ Pozostawienie części zmiennej w pliku list.blade.php oraz dodanie **@extends('template')** oraz ujęcie całej zawartości w **@sestion('content')**
	▪ W kontrolerze *ProductController* utworzenie tabeli z przykładowym produktami oraz dodanie parametru do *view* w metodzie index(). Tabela utworzona tymczasowo, później będzie podmieniona przez dane pobierane z przyszłej bazy danych. Póki co:
	`public function index() {`
	`$productsList = array(`
	`array("id"=>1,`
	`"name"=>"Produkt 1",`
	`"content"=>"Content about product 1",`
	`"amount"=>4,`
	`"price"=>100,`
	`"statusProduct"=>"avability"),` - dalej inne produkty wg zadania
	`);`
	`return view('products.list', ['productsList'=>$productsList]);`
	`}`
	▪ Dostosowanie tabeli w pliku list.blade.php do stworzonych powyżej danych z zastosowaniem pętli **@foreach($productsList as $product) ... @endforeach**

4. Stworzenie bazy danych
   a. Konfiguracja dostępu do bazy danych w zmiennych środowiskowych (plik .env)
   b. Utworzenie bazy danych o nazwie **bluebox** w phpMyAdmin
   c. Wykonanie pierwszej migracji z bazą danych
      Komenda: `php artisan migrate`

5. Stworzenie tabeli produktów
   a. Utworzenie tabeli o nazwie *products*
      Komenda: `php artisan make:migration add_products_table`
   b. Modyfikacja metody up() – dodania kolumn:

```
public function up()  {
    Schema::create('products',function(Blueprint $table){
        $table->id();
        $table->string('name');
        $table->string('content');
        $table->bigInteger('amount');
        $table->decimal('price');
        $table->string('statusProduct')->nullable();
        $table->timestamps();
    });
}
```

   c. Migracja z bazą danych
      Komenda: `php artisan migrate`

6. Modyfikacja tabeli użytkowników (w projekcie: istniejącej już tabeli *users*)
   a. Dodanie dwóch linijek kodu kolumn tabeli *users*, zlokalizowanej w katalogu migrations i nazwie ...create_user_table.php:
      - `$table->string('surname');` - dodanie nazwiska użytkownika
      - `$table->string('statusUser')->default('client');` - kolumna, określająca użytkownika aplikacji; czy jest to admin, czy klient? (wartość domyślna: klient)
   b. Migracja z bazą danych
      Komenda: `php artisan migrate`

   Uwaga: Istnieje możliwość dodania nowej migracji poprzez komendę php artisan make:migration add_users_details i ujęciu w metodzie up() nowych kolumn, a w metodzie down() ich cofnięcia z wykorzystaniem parametru dropColumn.

7. Wypełnienie tabeli *products* danymi stworzonymi poprzez tabelę Seeder
   a. Uzupełnienie tabeli *products* danymi o produktach z treści zadania z zastosowaniem tabeli Seeder (w późniejszym etapie dane te będą generowane w momencie dodania produktu do bazy)
   b. Utworzenie Seedera (katalog: database\seeder)
      Komenda: `php artisan make:seeder ProductsTableSeeder`
   c. W metodzie run() zastosowanie fasady DB (w tym, jej dodanie w części use) poprzez:

```
// PRODUCT 1
DB::table('products')->insert([
    'name'=>'Product 1',
    'content'=>'Content about product 1',
    'amount'=>4,
    'price'=>100,
    'statusProduct'=>'avability'
]);
```

      (...) dalej zgodnie z treścią zadania
   d. Migracja z bazą danych z opcją seed konkretnej klasy
      Komenda: `php artisan db:seed –class=ProductsTableSeeder`

Uwaga: Zastosowanie komendy php artisan migrate:fresh –seed poinformuje o pozytywnym wykonaniu migracji, lecz w aktualizowanej tabeli dane te się nie pojawią. W tym celu należy zmodyfikować metodę run() w pliku DatabaseSeeder.php i dodać: ***$this->call(ProductsTableSeeder::class);***

8. Wypełnienie tabeli *users* danymi stworzonymi poprzez tabelę Seeder

   a. Uzupełnienie tabeli *users* danymi użytkowników, określonych za pośrednictwem tabeli Seeder (w późniejszym etapie dane te będą generowane w momencie dodania użytkownika do bazy)

   b. Utworzenie Seedera (katalog: database\seeder)
   Komenda: `php artisan make:seeder UsersTableSeeder`

   c. W metodzie run() zastosowanie fasady DB (w tym, jej dodanie w części use) poprzez:

```
// ADMIN
DB::table('users')->insert([
    'name'=>'Marek',
    'surname'=>'Trocha',
    'email'=>'marek@trocha.net.pl',
    'password'=>bcrypt('123'),
    'statusUser'=>'admin',
]);
// CLIENT
DB::table('users')->insert([
    'name'=>'Adam',
    'surname'=>'Nowak',
    'email'=>'adam@nowak.pl',
    'password'=>bcrypt('asc'),
    'statusUser'=>'client',
]);
```

   d. Migracja z bazą danych z opcją seed konkretnej klasy
   Komenda: `php artisan db:seed –class=UsersTableSeeder`

Uwaga: Zastosowanie komendy php artisan migrate:fresh –seed poinformuje o pozytywnym wykonaniu migracji, lecz w aktualizowanej tabeli dane te się nie pojawią. W tym celu należy zmodyfikować metodę run() w pliku DatabaseSeeder.php i dodać: ***$this->call(UsersTableSeeder::class);***

9. Dodanie produktu

   a. W *ProductController.php* (w późniejszym etapie odseparowanie repozytorium od kontrolera) dodanie metody create(), odpowiedzialnej za dodawanie nowych produktów. Na tym etapie będą to dane statyczne, a później zostanie utworzony formularz dodawania nowego produktu. Po dodaniu nowego produktu, użytkownik zostaje przekierowany na stronę z produktami.

```
public function create() {
    Product::create([
        'name'=>'Product 20',
        'content'=>'Content about product 20',
        'amount'=>100,
        'price'=>199,
        'statusProduct'=>'avability'
    ]);

    return redirect('products');
}
```

   b. Utworzenie routingu (w pliku web.php):
   Route::get('products/create', 'App\Http\Controllers\ProductController@create');

   c. Tworzenie widoku – patrz punkt 19.

10. Edycja produktu

a. Utworzenie metody o nazwie edit($id), dzięki której będzie możliwość edycji produktu, wyszukanego po numerze id, stąd w argumencie funkcji identyfikator produktu. Na tym etapie będą to dane statyczne, a później zostanie utworzony formularz dodawania nowego produktu. Po dodaniu nowego produktu, użytkownik zostaje przekierowany na stronę z produktami.

```
public function edit($id) {
        $product = Product::find($id);
        $product->price = 9;
        $product->save();
        return redirect('products');
    }
```

b. Utworzenie routingu (w pliku web.php):
Route::get('products/edit/{id}, 'App\Http\Controllers\ProductController@edit);

11. Modyfikacja istniejącego modelu *User* oraz utworzenie modeli *Product* i *Income*

a. Modyfikacja istniejącego modelu *User* w katalogu Models poprzez dodanie w metodzie fillable() nazw kolumn ('surname' i' statusUser'), które mogą być zmieniane podczas aktualizacji

b. Utworzenie nowego modelu *Product* i dodanie metody fillable() z nazwami kolumn (jak powyżej): 'name', 'content', 'amount', 'price', 'statusProduct'.
Komenda: `php artisan make:model Product`

c. Utworzenie nowego modelu *Income* (służącego do kumulowania sprzedanych produktów) i dodanie metody fillable() z nazwami kolumn (jak powyżej): 'id_user, 'id_product,
Komenda: `php artisan make:model Income`

12. Modyfikacja kontrolera *ProductController*

a. Nadpisanie dotychczas statycznych danych w kontrolerze *ProductController.php* (patrz: punkt 3c) danymi z bazy danych:

- Dodanie use App\Models\Product;
- Nadpisanie wcześniej utworzonej tabeli kodem, odpowiedzialnym za pobranie danych z bazy, tj: **$products = Product::where('statusProduct','avability')->orderBy('id, 'asc')->get();** czyli pobranie produktów, których status jest ustawiony, jako dostępne (avability) i ułożenie ich rosnąco wg kolumny 'id. W tym przypadku wszystkie są dostępne, więc taki sam efekt dałoby wpisanie **$products = Product::all();**
- Dokonanie zmian w tabeli widoku resourses\views\products\list.blade.php

```
<th scope="row">{{ $product->id }}</th>
<td>{{ $product->name }}</td>
<td>{{ $product->content }}</td>
<td>{{ $product->amount }}</td>
<td>{{ $product->price }} PLN</td>
<td>{{ $product->statusProduct }}</td>
```

Uwaga: W tym miejscu można dodać odnośnik do pojedynczych produktów (w tym osobnego widoku) poprzez utworzenie nowej metody, np. o nazwie show($id), która będzie wynajdywała produkt po numerze id oraz utworzeniu nowego routingu, typu products/{id} z odwołaniem do utworzonej metody show($id).

13. Tworzenie wzorca Repository (BaseRepozytory.php)

Głównymi powodami oddzielenia modelu od kontrolera jest: przejrzystość kodu kontrolera oraz łatwość modyfikacji zapytań z bazy, jak i dowolnej zmiany samej bazy danych. Dlatego zostanie utworzona klasa pośrednicząca, która będzie pośredniczyć między modelem a kontrolerem. Tworzenie nowego folderu o nazwie *Repositories* w katalogu app, a w nim BaseRepository.php

```
<?php
namespace App\Repositories;
use Illuminate\Database\Eloquent\Model;
```

```
class BaseRepository {
    protected $model;

    // POBRANIE WSZYSTICH REKORDÓW Z BAZY
    public function getAll($columns = array('*')) {
        return $this->model->get($columns);
    }

    // TWORZENIE REKORDU W BAZIE ($noweDane)
    public function create($data) {
        return $this->model->create($data);
    }

    // AKTUALIZACJA REKORDU W BAZIE PO ID ($noweDane, $id)
    public function update($data, $id) {
        return $this->model->where("id",'=',$id)->update($data);
    }

    // USUWANIE REKORDU W BAZIE PO ID ($id)
    public function delete($id) {
        return $this->model->destroy($id);
    }

    // ZNAJDUWANIE REKORDU W BAZIE PO ID ($id)
    public function find($id) {
        return $this->model->find($id);
    }
}
```

14. Tworzenie Repozytorium ProductRepository.php, UserRepository.php i IncomeRepository.php

a. **ProductRepository** – obsługa modelu pod względem produktów. Tworzenie nowego pliku w katalogu *Repositories* o nazwie ProductRepository.php a w niej klasa dziedzicząca po klasie BaseRepository, dzięki czemu jest możliwość odwołania się do modelu produktów

```
<?php
namespace App\Repositories;
use App\Models\Product;

class ProductRepository extends BaseRepository {
    public function __construct(Product $model) {
        $this->model = $model;
    }
}
```

Warto dodać, że w powyższej klasie brakuje jednak odwołania do wyszukania produktu po jego dostępności oraz wyświetlenie wyniku wyszukiwania po np. numerze id produktu. Dlatego jest dodana nowa metoda o nazwie getAvability():

```
// POBRANIE REKORDU Z BAZY WG OKREŚLONEGO KRYTERIUM
public function getAvability() {
    return $this->model->where('statusProduct','avability')->
    orderBy('id','asc')->get();
}
```

Uwaga: Modyfikacja zapytań z bazy danych z zastosowaniem funkcjonalności Query Builder.
W pliku ProductReposiroty.php ciało metody getAvability() można zastąpić poprzez fasadę DB:
return DB::table('products')->where('statusProduct','=','avability')->get();

Ponadto, kontroler *ProductController.php* został dostosowany do wcześniej dokonanych zmian (szczegóły w pliku)

b. **UserRepository** – obsługa modelu pod względem użytkowników. Tworzenie nowego pliku w katalogu *Repositories* o nazwie UserRepository.php a w niej klasa dziedzicząca po klasie BaseRepository

```
<?php
namespace App\Repositories;
use App\Models\User;

class UserRepository extends BaseRepository {
    public function __construct(User $model) {
        $this->model = $model;
    }
```

c. **IncomeRepository** – obsługa modelu pod względem ujednolicenia wszystkich zakupionych produktów przez konkretnym użytkowników (klientów). Tworzenie nowego pliku w katalogu *Repositories* o nazwie IncomeRepository.php a w niej klasa dziedzicząca po klasie BaseRepository

```
<?php
namespace App\Repositories;
use App\Models\Income;

class IncomeRepository extends BaseRepository {
    public function __construct(Income $model) {
        $this->model = $model;
    }
```

15. Tworzenie tabeli *Incomes*

a. Utworzenie tabeli o nazwie *Incomes*

Komenda: `php artisan make:migration add_incomes_table`

b. Modyfikacja metody up() – dodania kolumn:

```
public function up()  {
    Schema::create('products',function(Blueprint $table){
        $table->id();
        $table->bigInteger('id_user');
        $table->bigInteger('id_product');
        $table->dateTime('date');
        $table->timestamps();
    });
}
```

c. Migracja z bazą danych

Komenda: `php artisan migrate`

16. Tworzenie kluczy głównych

a. Utworzenie tabeli z kluczem głównym

Komenda: `php artisan make:migration add_foreign_products_to_incomes`

b. W metodzie up():

```
Schema::table('incomes', function (Blueprint $table) {
   $table->bigInteger('id_product')->unsigned()->change();
   $table->foreign('id_product', 'incomes_id_product_foreign')->
   references('id')->on('products')->onDelete('cascade');
});
```

c. Utworzenie tabeli z kluczem głównym

Komenda: `php artisan make:migration add_foreign_users_to_incomes`

d. W metodzie up():

```
Schema::table('incomes', function (Blueprint $table) {
```

```
            $table->bigInteger('id_user')->unsigned()->change();
            $table->foreign('id_user', 'incomes_id_user_foreign')->
            references('id')->on('users')->onDelete('cascade');
        });
```

e. Migracja z bazą danych
   Komenda: `php artisan migrate`
f. Dla celów testowych, ręczne nadanie zmiennych (powiązań między użytkownikami, a kupionymi przez nich produktami) w tabeli *Incomes*

17. Tworzenie kontrolera i widoku dla części *Income,* pokazującej klientów i zakupione przez nich produkty)

a. Tworzenie kontrolera o nazwie IncomeController
   Komenda: `php artisan make:controller IncomeController`
b. W utworzonym kontrolerze dodanie metody index()

```
public function index(IncomeRepository $incomeRepo) {
    $incomes = $incomeRepo->getAll();

    return view('incomes.list', ['incomesList'=>$incomes,
                                 'footerYear'=>date("Y"),
                                 'title'=>'Customers and incomes']);
}
```

c. Dodanie routingu (w pliku web.php):
   Route::get(incomes/', 'App\Http\Controllers\IncomeController@index');
d. Tworzenie widoku (nowy katalog **incomes** i plik **list.blade.php)** oraz dostosowanie wyglądu i tabeli

18. Modyfikacja modelu *Income* (w katalogu Models) w celu wyświetlenia powiązań w widok

a. Dodanie dwóch metod user() oraz product()

```
public function user() {
    return $this->belongsTo(User::class,'id_user');
}
public function product() {
    return $this->belongsTo(Product::class,'id_product');
}
```

b. Dostosowanie zmian w widoku (resources\views\incomes\list.blade.php)

```
@foreach($incomes as $income)
<tr>
    <th scope="row">{{ $income->id }}</th>
    <td>{{ $income->user->name }} {{ $income->user->surname }}</td>
    <td>{{ $income->product->name }}</td>
    <td>{{ $income->product->price }} PLN</td>
    <td>{{ $income->date }}</td>
</tr>
@endforeach
```

19. Tworzenie widoku dla modułu dodawania produktu (patrz punkt 9)

a. Utworzenie nowego pliku create.blade.php (w katalogu resources\views\products\) z formularzem, a w nim powiązanie z kontrolerem *ProductController* i nową metodą **ProductController@store**, służącą do zapisywania zmian.
b. Utworzenie dwóch routingów:
   - Wyświetlenie strony edycji produktu:
     Route::get('products/create', 'App\Http\Controllers\ProductController@create');
   - Zapisanie zmian w edytowanym produkcie:
     Route::post('products/', 'App\Http\Controllers\ProductController@store');
c. W kontrolerze *ProductController* modyfikacja metody create() oraz dodanie nowej – store()
   - W metodzie create() nadpisanie statycznie wpisanego produktu 20 odwołaniem:
     return view('products.create');

- Utworzenie nowej metody o nazwie store()

```
public function store(Request $request) {
    $product = new Product;
    $product->name = $request->input('name');
    $product->save();

    return redirect()->action('ProductController@index');
}
```

d. W widoku resources\views\template.blade.php, w menu głównym strony dodanie nowej pozycji w menu „Products" o nazwie „Add product" i podlinkowanie do nowej strony products/create

20. Tworzenie widoku dla modułu edytowania istniejącego produktu (patrz punkt 10)

a. Tworzenie routingu (w pliku web.php) do wysyłania zaktualizowanych danych (routing do samego edytowania produktu został już stworzony podczas omawiania punktu 10b):
Route::post('products/edit/', 'App\Http\Controllers\ProductController@editStore');

b. W kontrolerze *ProductController* nadpisanie kodu ze statycznie wpisanymi nowymi danymi edytowanego produktu. Ostateczny kształt metody edit():

```
public function edit(ProductRepository $productRepo, $id) {
    $product = $productRepo->find($id);
    return view('products.edit', ['product'=>$product,
                                  'footerYear'=>date("Y")]);
}
```

c. Tworzenie widoku (nowy plik w resoursecs\views\products\edit.blade.php), związanego z edycją danych wybranego produktu oraz dodanie ukrytego pola input (w porównaniu do widoku create.blade.php), dzięki któremu będzie można „zlokalizować" edytowany produkt po id:
<input type="hidden" name="id" value="{{ $product->id }}" />

d. W widoku resources\views\products\list.blade.php dodanie linku (wg id produktu) z możliwością edycji danego produktu: **{{ URL::to('products/edit/' . $product->id) }}**

e. Utworzenie nowej metody w *ProductController* o nazwie editStore do wysyłania zmian:

```
public function editStore(Request $request) {
    $product = Product::find($request->input('id'));
    $product->name = $request->input('name');
    $product->content = $request->input('content');
    $product->amount = $request->input('amount');
    $product->price = $request->input('price');
    $product->statusProduct = $request->input('statusProduct');
    $product->save();

    return redirect()->action('App\Http\Controllers\ProductController@index');
    }
```

f. Walidacja formularza: w kontrolerze *ProductController* w metodzie store() dodanie walidacji:

```
$request->validate([
    'name' => 'required|max:255',
    'content' => 'required',
    'amount' => 'required',
    'price' => 'required'
]);
```

g. W widoku resources\views\products\create.blade.php dodanie elementu, odpowiedzialnego za wyświetlenie błędu, związanego z walidacją formularza – w @section('content'):

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
```

```
                    </ul>
                </div>
            @endif
```

21. Usuwanie produktu

a. W widoku resources\views\products\list.blade.php dodanie linku (wg id produktu) z dodatkową opcją potwierdzenia chęci usunięcia produktu:
{{ URL::to('products/delete/'. $product->id) }} onclick="return confirm('Delete the product?')"

b. Dodanie routingu (w pliku web.php)
Route::get('products/delete/{id}', 'App\Http\Controllers\ProductController@delete');

c. Modyfikacja kontrolera i tworzenie metody delete():

```
public function delete(ProductRepository $productRepo, $id) {
    $product = $productRepo->delete($id);
    return redirect('products');
}
```

22. Logowanie do systemu

a. Przekopiowanie panelu logowania i rejestracji, czyli części kodu z pliku resources\views\layouts\app.blade.php (nadanego przez framework Laravel) z obszaru <!-- Right Side Of Navbar -->, tj. od 40 do 73 linijki do stworzonego już pliku widok projektu – template.blade.php

b. Przeniesienie części menu głównego do obszaru dostępnego tylko po zalogowaniu:
@guest (..) Logowanie i rejestracja **@else** zawartość po zalogowaniu **@endguest**

c. Dostosowanie layoutu widoku logowania i rejestracji do wyglądu realizowanego projektu
- W pliku: resources\views\auth**login**.blade.php
  zmiana ścieżki w części @extends na 'template' oraz drobne poprawy wizualne widoku
- W pliku: resources\views\auth**register**.blade.php
  zmiana ścieżki w części @extends na 'template' oraz drobne poprawy wizualne widoku
- W pliku: resources\views**home**.blade.php
  zmiana ścieżki w części @extends na 'template' oraz drobne poprawy wizualne widoku
- W pliku: resources\views\passwords**reset**.blade.php
  zmiana ścieżki w części @extends na 'template' oraz drobne poprawy wizualne widoku
- W pliku: resources\views\passwords**email**.blade.php
  zmiana ścieżki w części @extends na 'template' oraz drobne poprawy wizualne widoku

23. Zabezpieczenie zasobów (middleware)

Na tym etapie nastąpi zabezpieczenie tych widoków, które powinny być widoczne tylko dla zalogowanych użytkowników (aby osoby niezalogowane nie mogły mieć dostępu do zabezpieczonych widoków poprzez ręczne wpisanie adresu strony w pasku URL). Dotyczy to dwóch kontrolerów: *ProductController* oraz *IncomeController*.

ProductController

a. Dodanie fasady **use Illuminate\Support\Facades\Auth;**

b. Dodanie metody:

```
public function __construct() {
    $this->middleware('auth');
}
```

c. W poszczególnych metodach konstruktora *ProductController*:
- index() – wyświetlenie widoku (tylko dla klientów i admina)

```
if (Auth::user()->statusUser != 'client' &&
    Auth::user()->statusUser != 'admin') {
return redirect()->route('login');
}
```

- create() – tworzenie produktu (tylko dla admina) – jak powyżej, ale tylko `(Auth::user()->statusUser != 'admin')`
- store() – wysyłka stworzonego produktu do bazy (tylko dla admina) – jak powyżej
- edit() – edycja produktu (tylko dla admina) – jak powyżej
- editStore() – wysyłka edytowanego produktu do bazy (tylko dla admina) – jak powyżej
- delete() – usuwanie produktu (tylko dla admina) – jak powyżej

IncomeController

d. Dodanie fasady **use Illuminate\Support\Facades\Auth;**

e. Dodanie metody:

```
public function __construct() {
    $this->middleware('auth');
}
```

f. W poszczególnych metodach konstruktora *ProductController*:

- index() – wyświetlenie widoku (tylko dla klientów i admina)

```
if (Auth::user()->statusUser != 'admin') {
return redirect()->route('login');
}
```

g. W widoku template.blade.php dostosowanie zmian widoku menu głównego w zależności od typu użytkownika: czy jest to użytkownik (klient) czy administrator systemu.

Uwaga: Powyższe zabezpieczenie middleware można również zastosować w **routingu**, gdzie przy wybranych ścieżkach dodajemy **(...)->middleware('auth');.**

24. Rejestracja użytkownika (klienta) oraz moduł „Przypomnienie hasła"

Dodanie użytkownika (klienta)

a. W formularzu rejestracji (resources\views\auth\register.blade.php) w znaczniku <form> odwołanie się do metody store (jej tworzenie poniżej) w kontrolerze *UserController* i dodanie: **action="{{ action('UserController@store') }}"**

b. Dodanie routingu: **Route::post('users/', 'App\Http\Controllers\UserController@store');**

c. Utworzenie metody store w kontrolerze *UserController*

```
public function store(Request $request) {

    $request->validate([
        'name' => 'required|max:255',
        'surname' => 'required',
        'email' => 'required',
        'password' => 'required'
    ]);

    $user = new User;
    $user->name = $request->input('name');
    $user->surname = $request->input('surname');
    $user->email = $request->input('email');
    $user->password = bcrypt($request->input('password'));
    $user->save();

    return view('users.confirm', ['footerYear'=>date("Y"),
                                  'title'=>'Customers']);
}
```

d. Utworzenie widoku z podziękowaniem za rejestrację (resources\views\users**confirm**.blade.php)

Przypomnienie hasła

e. Po dokonaniu zmian w widokach reset.blade.php, email.blade.php oraz confirm.blade.php nastąpi przekierowanie do skrzynki pocztowej, której parametry są ujęte w pliku ze zmiennymi środowiskowymi (plik o nazwie .env). Dla celów niniejszego zadania etap ten zostanie pominięty.

25. Podgląd produktów zakupionych przez użytkownika (klienta)

a. Utworzenie routingi (w pliku web.php) odpowiedzialnego za trasę widoku, odpowiedzialnego za wyświetlenie użytkownikowi (klientowi) listy zakupionych produktów.
Route::get('incomes/show', 'App\Http\Controllers\IncomeController@show');

b. W kontrolerze *IncomeController* utworzenie nowej metody show(), odpowiedzialnej za odebranie wyniku wyszukania z bazy (punkt poniżej) i przekazanie jego przekazanie do widoku

```
public function show(IncomeRepository $productRepo) {

    $products = $productRepo->getYourProducts();
    return view('incomes.show', ['products'=>$products,
                                 'footerYear'=>date("Y"),
                                 'title'=>'Your products']);
}
```

c. W repozytorium *IncomeRepository* dodanie metody getYourProducts(), odpowiedzialnej za wyszukanie produktów użytkownika (klienta)

```
public function getYourProducts() {
    return
       DB::table('incomes')
      ->where('id_user','=', (Auth::user()->id))
      ->get();
}
```

Utworzenie widoku (resources\views\incomes**show.blade.php**)

26. Zakup produktów przez użytkowników (klientów)

a. Utworzenie routingów (w pliku web.php):
- Routing odpowiedzialny za wyszukanie kupowanego produktu po jego numerze id;
Route::get('products/buy/{id}', 'App\Http\Controllers\ProductController@buy');
- Routing odpowiedzialny za zapisanie zmian (kupienie produktu), tj. dodanie produktu do koszyka użytkownika (klienta) oraz aktualizacja bieżącego stanu magazynowego produktu (pamiętając o tym, aby ilość produktów w magazynie nie była niższa niż zero;
Route::post('products/store/', 'App\Http\Controllers\ProductController@buystore');

b. W kontrolerze *ProductController* utworzenie dwóch metod:
- Metoda buy() odpowiedzialna za wyszukanie kupowanego produktu po jego numerze id;

```
public function buy(ProductRepository $productRepo, $id) {

    if (Auth::user()->statusUser != 'client' &&
        Auth::user()->statusUser != 'admin') {
        return redirect()->route('login');
    }

    $product = $productRepo->find($id);
        return view('products.buy', ['productBuy'=>$product,
                                     'footerYear'=>date("Y")]);
}
```

- Metoda buyStore() odpowiedzialna za pośrednictwo pomiędzy bazą a widokiem;

```
    public function buyStore(Request $request) {

        if (Auth::user()->statusUser != 'client' &&
            Auth::user()->statusUser != 'admin') {
            return redirect()->route('login');
        }

        ProductRepository::buyStore($request->input('id'));

            return redirect()->action
            ('App\Http\Controllers\ProductController@index');
    }
```

c. W repozytorium *ProductRepository* utworzenie metody, odpowiedzialnej za bezpośredni kontakt z bazą danych. Metoda dodaje zakupiony przez klienta produkt do jego koszyka produktów, a jednocześnie aktualizuje tabelę produktów w pozycji ilości dostępnych produktów.

```
static public function buyStore($id_product) {

    $id_user =  Auth::user()->id;
    $amount = DB::table('products')
                    ->where('id', $id_product)
                    ->value('amount');

    if ($amount <= 0) {
        return $amount = 0;
    } else {
        $newAmount = $amount - 1;
    }

    return (DB::table('incomes')
         ->insert(['id_user' => $id_user,
                   'id_product' => $id_product,
                   'created_at' => now() ]))
                    &&
           (DB::table('products')
         ->where('id', '=', $id_product)
         ->update(['amount' => $newAmount ]));
}
```

d. Utworzenie widoku (resousces\views\products**buy.blade.php**), odpowiedzialnego za wyświetlenie zawartości produktu, który jest kupowany i przeprocesowanie użytkownika (klienta) przez procedurę kupna produktu. W tym miejscu można zaimplementować moduł odpowiedzialny za realizację płatności za produkt. W niniejszym zadaniu proces ten zostaje pominięty (utworzenie widoku roboczego: **payment.blade.php** powiązanego z metodą payment() w *ProductController*.

27. Dodanie kryteriów wyszukiwania (z treści zadania)

a. W repozytorium *ProductRepository* dodanie nowych metod:
   - getAvability() – wyświetlenie produktów dostępnych,
   - getUnavability() – wyświetlenie produktów dostępnych,
   - amountZero – wyświetlenie produktów mających 0 (zero) na stanie
   - amountFive – wyświetlenie produktów mających więcej niż 5 sztuk na stanie

b. W kontrolerze *ProductRepository* dodanie metod, analogicznych do punktu powyżej; przykład:

```
    public function indexAvability(ProductRepository $productRepo) {

        if (Auth::user()->statusUser != 'client' &&
           Auth::user()->statusUser != 'admin') {
```

```
                return redirect()->route('login');
            }

            $products = $productRepo->getAvability();
            return view('products.list', ['productsList'=>$products,
                                          'footerYear'=>date("Y"),
                                          'title'=>'Products']);
        }
```

c. Dodanie list rozwijanych z dostępnymi opcjami przy kolumnie „Amount" oraz „Status of product" (w pliku: resources\views\products\list.blade.php) i dodanie do nich tras z punktu poniżej

d. Dodanie routingów (w pliku web.php):

- Route::get('products/avability', 'App\Http\Controllers\ProductController@indexAvability');
- Route::get('products/unavability', 'App\Http\Controllers\ProductController@indexUnavability');
- Route::get('products/zero', 'App\Http\Controllers\ProductController@amountZero');
- Route::get('products/more', 'App\Http\Controllers\ProductController@amountFive');

28. Wizualne poprawienie widoku tabeli z produktami

a. **Usunięcie z widoku produktów przycisku „Buy / download" przy zerowym stanie produktów lub przy produkcie niedostępnym**

W pliku widoku (resources\views\products\list.blade.php) dodanie instrukcji, odpowiedzialnej za zniknięcie przycisku „Buy / download" przy zerowym stanie produktów.

```
@if (($product->amount != 0) && ($product->statusProduct == 'Avability'))
 <form class="form-inline">
   <a href="{{ URL::to('products/buy/' . $product->id) }}">
     <button class="btn btn-sm btn-outline-secondary" (…)
   </a>
 </form>
@endif
```

b. **Usunięcie stanu produktów w przypadku produktów niedostępnych**

W pliku jak powyżej dodanie:

```
@if ($product->statusProduct == 'Avability')
   {{ $product->amount }}
@endif
```

29. API, czyli dostęp do systemu dla zewnętrznych aplikacji

Tworzenie API na przykładzie dostępu do informacji o produktach

a. Tworzenie nowego folderu o nazwie **Api** w katalogu App\Http\Controllers

b. Tworzenie nowego pliku ProductController.php i jego dostosowanie na przykładzie pliku *Product-Controller*, tj. zmiana przestrzeni nazw, usunięcie zbędnych metod (pozostawienie metody index()) oraz dodanie **use Illuminate\Routing\Controller as BaseController;** + modyfikacja nazwy klasy bazowej z (...) extends Controller na (...) **extends BaseController**

c. Zwrócenie listy produktów w formacie JSON z wykorzystaniem metody **toJson()**.

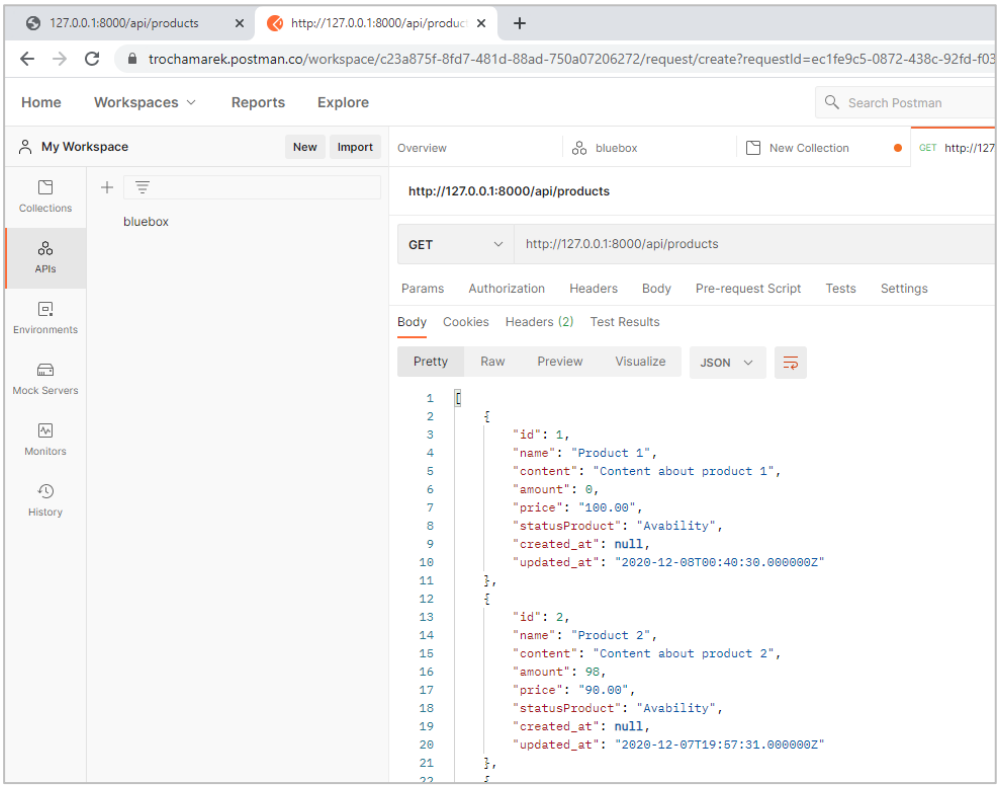
Cały plik *Api\ProductController.php* wygląda następująco:

```
<?php
namespace App\Http\Controllers\Api;

use App\Repositories\ProductRepository;
use Illuminate\Routing\Controller as BaseController;

class ProductController extends BaseController {
        public function index(ProductRepository $productRepo) {
```

```
                $products = $productRepo->getAll();
                return $products->toJson();
            }
        }
```

d. Dodanie routingu, ale już w katalogu routes\api.php (a nie web.php):
Route::get('products/', 'App\Http\Controllers\Api\ProductController@index');

e. Testowanie API z wykorzystaniem narzędzia Postman. Screen z przykładowym testem Api\products:



Powyższy opis można również zastosować do stworzenia API dla np. dostępu do użytkowników serwisu (*UserController)* czy zakupionych produktów (*IncomeController*). Co ważne:

- dzięki odseparowaniu repozytorium można tu zastosować API, jak i np. blade, bez konieczności in-gerencji w same zapytania do naszej bazy danych;
- w samym routingu API można również stosować inne typy zapytań niż get, np: post.

30. Zakończenie

a. Reset bazy danych i ich zawartości (danych ujętych w seederach) do założeń niniejszego zadania
Komenda: `php artisan migrate fresh --seed`

b. Po zakończeniu zarówno części developerskiej, jak i testowej można rozpocząć uruchomienie serwisu w części produkcyjnej z wcześniejszym uwzględnieniem zmian zmiennych środowiskowych w pliku .env, np.: APP_ENV=production, APP_DEBUG=false, APP_URL=www.nazwadomeny.pl/bluebox, edycja danych dostępowych do produkcyjnej bazy danych, ustawienia skrzynki pocztowej, Itp...

Marek Trocha
tel. 693 877 101
e-mail: marek@trocha.net.pl
strona: trocha.net.pl